A new approach to software development for embedded control systems

Vaclav Cechticky, Alessandro Pasetti, Walter Schaufelberger

Automatic Control Laboratory, Physikstrasse 3, CH-8092 Zürich, Switzerland
{cechti,pasetti,ws}@aut.ee.ethz.ch

This paper presents an innovative approach to the development of the software for embedded control systems that is being investigated at the Automatic Control Laboratory (Institut für Automatik or IfA) at ETH-Zurich. The approach is based on two planks: real-time Java as a programming language and component-based software frameworks.

1 INTRODUCTION

Current embedded control systems are typically implemented in C or Ada. The motivation for switching to Java is the greater safety of this language, its support for multi-tasking and networking operations, and the lower costs that derive from its wider user base. Despite these advantages, Java cannot at present be used in embedded systems because it lacks support for hard real-time operations and for low-level interactions with hardware. These shortcomings are in the process of being addressed: some suppliers are already providing implementations of Java that are RT-compliant and a RT extension of the language has recently been approved by Sun Microsystems.

Against this background, the paper discusses the work being done at IfA to evaluate the use of RT Java for embedded control systems. More specifically, the results of a project done for the European Space Agency to port a prototype software framework for satellite control systems to Java are presented [11].

The paper begins with an overview of framework technology, which we regard as the cornerstone technology for the development of embedded control software. This is followed, in section 3, by a discussion of the suitability of Java as an implementation language for software frameworks for real-time systems. Sections 4 and 5 present the results of the porting exercise of the satellite control framework from C++ to RT Java. Finally, section 6 gives an overview current work in IfA that concentrates on the instantiation process for software frameworks for embedded control systems written in Java.

2 SOFTWARE FRAMEWORKS

The reuse potential promised by object-orientation finds its fullest realization in software frameworks. Software frameworks are a software reuse technology that promotes the reuse of an entire architecture (as opposed to just code fragments). A software framework provides a generic architectural skeleton from which applications within a specific domain can be rapidly instantiated. Its typical representation is shown in Figure 1. The unshaded area represents the architectural backbone shared by all applications in the framework domain. The framework captures this architectural backbone and makes it available to application developers who tailor it to their needs by plugging into the framework components that implement the application-specific behaviours (the darker boxes in the Figure 1).

Software frameworks go beyond subroutine or class libraries because they make available not just individual modules but also the relationships between them. It is in this sense that they allow reuse of architecture as well as of code. Subroutine and class libraries, however, are generic artifacts that can be used in a large variety of applications whereas frameworks are targeted at a specific – and often narrow – domain. They aim at depth rather than breadth of reuse.

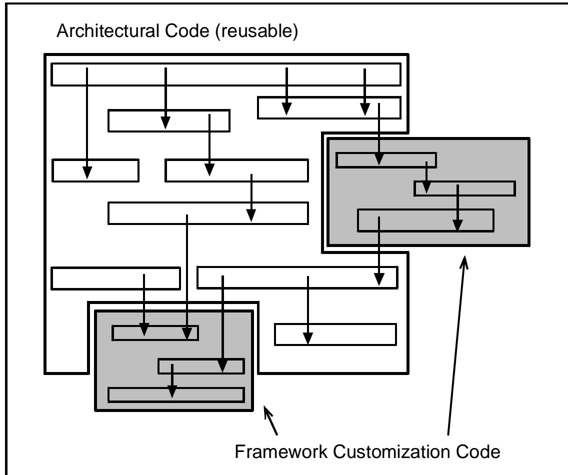

Figure 1. Representation of a software framework as customizable architecture.

2.1 The AOCS Framework

The AOCS Framework is one particular representative software framework that was developed as an object-oriented reusable architecture for on-board Attitude and Orbit Control Systems (AOCS) by one of the authors. The AOCS Framework facilitates the development of on-board control applications. It is constructed as a set of plug-compatible components. The framework components are adaptable to allow easy matching of application requirements. The adaptation mechanisms are object-composition and inheritance [3,10].

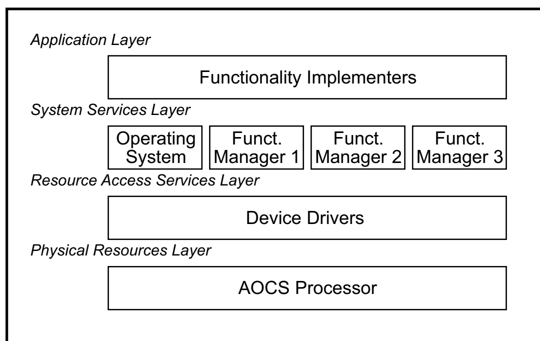

Figure 2. Structure of Applications Instantiated from the AOCS Framework

The structure of an application instantiated from the AOCS Framework is shown in Figure 2. The framework provides a set of "functionality managers" that are best seen as domain-specific extensions of the operating systems. The functionality managers are invariant within the target domain of the framework and are fully reusable. The functionality managers are customized to match the requirements of an individual application by combining them with application-specific components that form the top layer of the application. Because of its heavy use of object-oriented constructs, the AOCS Framework is regarded as typical of the kind of on-board applications that would be built in Java.

3 JAVA – THE LANGUAGE FOR SOFTWARE FRAMEWORKS

Object-oriented software frameworks can be implemented in any object-oriented language but, among mainstream languages, Java is the most well-suited for this purpose for two reasons. Firstly, Java is a fully object-oriented language, which forces designers to build their applications in an object-oriented way. Secondly, Java offers a dedicated construct to represent abstract interfaces and abstract interfaces play a crucial role in framework design because the define the interface between the architectural skeleton provided by the framework and the customization components that are used to instantiate it (in term of the metaphor of Figure 1, abstract interfaces define the "shape" of the "adaptation holes" and therefore define which components can fill which holes).

Java is also interesting for other reasons. Like Ada, it is designed to promote safe coding practices and it has built-in support for tasking and networking operations. It is superior to Ada because it leads to higher development productivity, it is more portable, and its wide user base leads to lower development costs. The availability of automatic documentation facilities within the language is another asset of Java, which is especially important in mission-critical domains. The disadvantages of Java are its high memory and CPU requirements, its unsuitability for real-time applications and its lack of certification. These problems however are being addressed. Dynamic compilation techniques can make Java as fast as conventional languages. Java virtual machines targeted at the embedded market have very small memory footprint. Finally, a speci-

fication for a real-time extension of Java has been released and several suppliers have announced their intention of implementing real-time compliant Java virtual machines.

The Java language was designed to enforce the "write-once-run-anywhere" paradigm or WORA. The need to preserve compatibility with a wide variety of platforms and operating systems severely constrained the specification of the concurrency constructs. Their implementation within the JVM has to rely on services offered by the underlying operating system and the diversity of commercial operating systems is such that ensuring compatibility with all of them inevitably resulted in "weak" specification at Java language level. In fact, many operating systems in common usage – most notably the older versions of Unix – simply do not support real-time operation and would have been a very poor basis upon which to build RT-enabled Java virtual machine. Java applications were furthermore intended to run in a desktop/workstation environment shared with other (non-real-time) applications. Coexistence of real-time and non-real-time applications can be problematic and this too militated against giving Java real-time capabilities. These constraints resulted in a language that, when seen from the point of view of a real-time programmer, looks under-specified. Most of the basic syntactical constructs that one normally associates with real-time programming – in particular tasking, synchronization and event handling constructs – are present but their semantics is not strong enough to build programs with the level of determinism and timing predictability that is essential in the hard real-time world.

Additionally, real-time applications need a close interaction with the hardware than is common in desktop applications – arguably the natural target for Java. Java is deficient in this respect too and its deficiency is again a consequence of its virtual machine model. In order to preserve WORA, Java applications are supposed to interact only with the JVM and the language does not offer facilities to directly manipulate memory, hardware registers or interrupt vectors. As a consequence, interrupt handlers and device drivers – basic elements of most real-time applications – are simply impossible be develop in Java.

Finally, the presence of the garbage collector and the dynamic nature of the language make static analysis difficult or impossible.

In the Table 1 are summarized the main shortcomings of Java for real-time applications that have been identified during the research project [11].

Issue	Problem
Threading Model	Under-Specification: no guarantees about dispatching policy of ready threads
Intra-Thread Synchronization	Missing Feature: no safe means to suspend a thread until a well-defined time in the future
Inter-Thread Synchronization	Under-Specification: wait and notify facilities are present but order of release of notified threads is not specified
Access to Shared Resources	Under-Specification: no priority inversion policy is specified for synchronized statement
Memory Allocation	Under-Specification: memory is allocated from the heap but allocation policy is not specified and could be non-predictable and non-deterministic
Garbage Collection	Under-Specification: gc collection algorithm and its triggering conditions are not specified and could be non-predictable and non-deterministic
HW Interrupt Handling	Missing Feature: no facilities for linking threads or event handlers to hw interrupts
Asynchronous Event Handling	Under-Specified: standard Java event model adequate if threading model were adequate
Dynamic Class Loading	Incompatibility: dynamic class loading disrupts timing behaviour
Class Initialization	Under-Specification: no guarantee about the when classes are initialized

Table 1. Summary of shortcomings of standard Java for RT systems

3.1 Implementation road to Real-Time Java

One solution to the problem of making Java real-time is simply to develop a Java virtual machine that implements the real-time constructs of the Java language in a manner that allows timing predictabili-

ty analyses to be performed and hence hard real-time applications to be developed.

Consider for instance the problem of thread scheduling. The Java language does not give any guarantee about how threads of different priorities are scheduled. However, the language does not *forbid* any particular scheduling policy. A JVM developer is therefore free to provide an implementation that enforces a scheduling policy that is suitable for hard real-time applications. The same approach can be followed with all other identified shortcomings summarized in the Table 1. The result would be a JVM that is compliant with the language specification but that offers some additional guarantees – possibly sufficient to develop hard real-time applications.

Four suppliers are offering RT-compliant implementations of the JVM. NewMonics offers PERC [8], which is arguably the best-established solution. Esmertec offers a rival product called JBed with reduced functionalities and shorter heritage [4], and Aicas its JamaicaVM [1]. Finally, aJile took a different route and developed chip that implements a RT-compatible JVM in hardware [2]. This means that with the aJile concept, Java bytecodes are treated as a kind of assembler language and are directly interpreted at hardware level.

These solutions to the real-time Java problem arise from individual suppliers deciding to provide more or less idiosyncratic implementations of the Java language – possibly with additional support packages – designed to make development of real-time programs possible.

3.2 Specification road to Real-Time Java

An alternative route to the Real-Time Java is based on a formal definition of an extension of the language to be agreed by a community of users. The first milestone on this road was laid in 1999 by the National Institute of Standards and Technology (NIST) that prepared a report on the "Requirements for Real-Time Extension for the Java Platform" [9]. This document became the basis for two parallel efforts to produce a specification for real-time extension of Java. One was carried out under the aegis of Sun's Java Community Process [12]. The output is the so-called RTSJ (standing for "Real Time Specifications for Java"), which were approved as a formal extension to the Java language in November 2001. The second set of specifications are being elaborated by the J Consortium that brings together a group industry stakeholders that are independent of Sun Microsystems [7]. The output is the "Core Java" specifications.

Despite being inspired by the same set of high-level requirements, Core Java and the RTSJ have very different characteristics. The design of the former was driven by the desire to allow the development of real-time applications with speed of execution, latency and memory footprint comparable to those achievable in current C and C++ RTOS-based programs. The pursuit of run-time efficiency and small memory footprint led the J Consortium to develop a model of Java that is at odds with that used in the rest of the Java community. Their Core Java is intended to be a compiled language and compilation is not to bytecodes but directly to native code. Communications with Java applications running on conventional JVMs is possible through dedicated gateways. The Core Java execution environment is therefore best seen as a plug-in module to a standard JVM but Core Java also allows the development of stand-alone and highly optimized applications. The intention behind Core Java seems to be to allow programmers of hard real-time systems operating in memory- and CPU-constrained environment to develop their software using a Java-like syntax. As a result, the general language model behind Core Java is rather different from that of standard Java. The most notable difference is the lack of garbage collection which again shifts responsibility for managing memory to the programmer. For this an other reasons, Core Java cannot guarantee backward compatibility (i.e. there is no guarantee that a standard Java program will execute correctly in a Core Java environment). The RTSJ designers instead explicitly aimed at compatibility with the standard Java model. Their specifications center around the definition of APIs to allow real-time programming and their implementation implies a JVM with special capabilities to support them but otherwise identical to a conventional JVM. Both RT and non-RT threads are assumed to run within the same virtual machine. The most obvious price that the RTSJ pay for their adherence to the JVM model is run-time and memory penalties. Their version of RT Java – unlike that of the J Consortium – will not be able to compete with conventional real-time applications built around RTOSs. They will however be ideally suited for large-scale soft and

hard real-time applications where there is a concern for consistency with standard Java and a desire to use standard tools and development environments.

In another difference, the RTSJ are designed to be "open" with respect to implementation. They define interfaces and their semantics but do not mandate specific algorithms to implement them. Rather, they define a framework within which such algorithms can be defined by the developers. Where appropriate, sensible default choices are predefined. For instance, the specifications do not prescribe any scheduling algorithm but instead define a mechanism through which users can load components implementing their own scheduling algorithms. In view of its popularity, fixed priority scheduling is however provided as default scheduling algorithm. In the interest of simplicity and compactness, Core Java tends to be more specific and gives less leeway to users and future implementers.

A further crucial difference concerns the licensing model. The RTSJ were developed by an industrial consortium that had the backing of Sun Microsystems and that operated under the auspices of the Java Community Process [Sun98]. Their specifications are subject to the same licensing agreement as standard Java.

At the time of writing, the RTSJ are available in final form whereas the Core Java specifications are still in draft form. Additionally, there is one reference implementation for the RTSJ specification (from TimeSys) but none yet for Core Java.

Most observers believe that the RTSJ version of RT Java will prevail and will secure the largest presence in the RT Java market. There are at least three factors that point towards this conclusion. First and foremost, the RTSJ are better suited for soft real-time applications that will probably form the largest segment in the RT Java market. Secondly, they have the backing of Sun Microsystems and are fully integrated in the standard Java model. Finally, they are at a more advanced stage of development: more suppliers have announced support for the RTSJ than for Core Java.

4 PORTING THE AOCS FRAMEWORK TO REAL-TIME JAVA

A full assessment of the suitability of Java for real-time applications will have to wait for the RT Java specifications to stabilize and for reliable compliant implementations to emerge. In the meantime, as a way of performing a preliminary assessment, an effort was made to port the AOCS Framework to a version of real-time Java [11].

The porting exercise used as a target Esmertec's JBed running on a RPX Lite board mounting a PowerPC 823 microprocessor. The Esmertec JVM was selected on the basis of a preliminary analysis of its suitability and because it is supplied by a Zürich-based company. However, in recognition of the variety of RT Java solutions currently on the market and of the uncertainty about the one that will eventually prevail, the decision was taken to make the Java version of the framework independent of the RT aspects of the language. This was achieved by writing the code for all framework components in standard Java (i.e. no use was made of constructs or services that are specific to the real-time extension of the language or to the Jbed platform). Hence, the framework, although tested on JBed, is compatible with other real-time version of Java.

The Jbed platform offers a nearly complete support of the JDK 1.2 APIs and adds to it some guarantees about real-time behaviour. In particular, it strengthens the Java threading model. Jbed schedules threads using a priority-based scheduling policy with a priority inheritance mechanism to avoid unbounded priority inversion. The more robust ceiling priority protocol is not provided. In addition to standard threads, Jbed provides a custom class `Task` to encapsulate threads with more flexible scheduling and task synchronization policies. In its current implementation, Task supports Earliest Deadline First scheduling but also allows the definition of periodic tasks. Class Task is integrated in the Java thread model because it is obtained by subclassing Thread.

Jbed provides an incremental automatic garbage collector that is scheduled as a low-priority background task. If a real-time task makes a request for memory that cannot be satisfied, the garbage collector is activated with the priority of the blocked task.

Additionally, Jbed provides a very good framework for developing device drivers in Java and for interfacing to the hardware. In particular, it is possible to link release of a task to reception of a specific interrupt.

Experience with the Jbed platform in the AOCS Framework porting project has been mixed. Two major problems stand out. Firstly, the AOCS Fra-

mework adheres to a policy of never using dynamic memory allocation in its RT part. This policy is in place to avoid dependencies on garbage collector implementations that not all RT java implementations provide. The Jbed documentation however does not document usage of dynamic memory allocation in the Java libraries and run-time services. Strict adherence to the policy of no dynamic memory allocation is therefore impossible.

Secondly, the AOCS is an inherently cyclical system and most of its functionalities are usually implemented as periodic tasks. Additionally, in an AOCS there is a requirement that sensors and actuators be sampled at specific times within a cycle and hence the periodic tasks that control the sampling must be exactly phased Jbed supports periodic tasks but does not have a sleep_until facilities. This makes it impossible to control accurately the phase of tasks.

5 REAL-TIME JAVA FRAMEWORK TEST

A framework is tested by using it to instantiate a prototype application within its domain. The RT Java Framework was tested by using it to instantiate a simple control system for a Swing Mass Model (SMM). The SMM is standard laboratory equipment for testing control algorithms with hardware-in-the-loop. It consists of two rotating disks connected with a torsional spring. The objective of the control action is to control the speed and angular position of the right-hand disk by acting on the left-hand disk. The system actuator is the electric motor that drives the left-hand disk and that can control both its position and its angular speed. The system sensors are position and angular speed sensors on the right-hand disk. A second motor is available that can apply a torque to the right-hand disk to simulate the presence of a disturbance torque. The SMM apparatus is shown in Figure 3.

In the configuration used for the instantiation of the RT Java Framework, the SMM is used as a single degree of freedom system where the angular speed of the right-hand motor is used to control the speed of the left-hand disk. The left-hand motor is not used.

Figure 3. Swing Mass Model Apparatus

The application instantiated from the AOCS Framework to control the SMM has the following features:

- Two operational modes
- Failure detection checks on the main system variables
- Failure recovery actions autonomously executed upon detection of failures
- Provision of telemetry data in two different and alternating formats
- Processing of four user telecommands
- Capability to perform manoeuvres (activated by telecommand) to force a profile on the control set-point

The total number of classes in the prototype application is 421. This includes the classes making up the JVM run-time system. The application-specific classes are just over 200. The prototype instantiation was based on four RT threads with the following functions:

- Thread 1: periodic thread to collect and process sensor measurements (including implementation of failure detection and recovery algorithms and of control laws)
- Thread 2: periodic thread to send commands to the actuators
- Thread 3: periodic thread to send TM data to a ground station simulator
- Thread 4: periodic thread that checks whether any telecommands have been received and, if so, loads them into the telecommand manager

The period was in all cases 1 second corresponding to the control cycle of the target application. The first two threads run at very high priority and cannot be pre-empted. They only perform RT-safe operations. The bottom two threads perform RT-unsafe operations (in particular, they use dynamic memory allocation). For this reason, they run at lower priority and can be pre-empted by the first two threads.

The memory requirement for the prototype application is 522 kBytes (data+code). Of these, less than 200 kBytes are for the Java run-time system. Note that, in the Jbed execution model, the Java class files are compiled to native code and linked to the JVM. The linker only includes classes and code that are actually referred to. These memory figures are therefore highly optimised.

Typical timing requirements for the execution of one cycle of the four above threads are:

- Thread 1: 10-26 ms, depending on operational conditions
- Thread 2: 1 ms
- Thread 3: 10-17 ms, depending on amount of TM data
- Thread 4: 1 ms

The processor was PowerPC 823 and the measurements were made when it was running at a frequency of 66 MHz. Note that there are no tools for computing the worst case execution times of a Java applications and the above values were derived from observation of a large number of measurements.

6 BEAN-BASED AUTOMATED INSTANTIATION ENVIRONMENT

Software frameworks are a software reuse technology that promotes the reuse of an entire architecture within a narrow domain. A software framework is an artefact that models the commonalities of all applications within a certain domain and that is designed to be easily transformed – or, to use the technical term, *instantiated* – into a specific application in that domain.

JavaBeans is the component standard of the Java language. A *Java bean* is a software component that can be manipulated graphically in so-called *beanbox tools* [13]. A beanbox tool is an environment where an application can be assembled by configuring and linking individual bean component. The configuration and linking code is automatically generated by the beanbox.

Software frameworks and beanboxes are complementary technologies because the framework can provide the components to be assembled within the beanbox environment and the rules for assembling them. Speaking figuratively, a beanbox is an empty container for which the framework supplies content. Together, framework and beanbox create an environment where applications within the framework's domain can be easily and rapidly instantiated. This type of environment is called an automated instantiation environment. Automated instantiation environments can be seen as customizable autocoding tools.

Automated instantiation tools have been successfully applied to many non real-time domains where they have led to enormous productivity gains. We are investigating their use for real-time embedded applications and we demonstrate their applicability by developing a *prototype automated instantiation environment.* The principle such an automated instantiation environment is sketched in Figure 4.

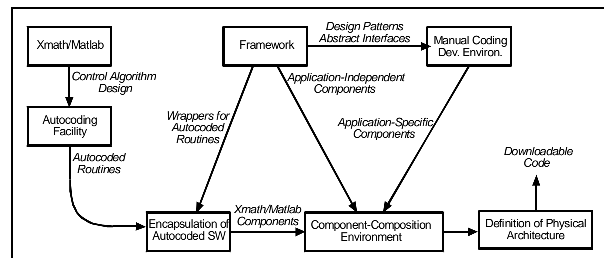

Figure 4. Automated Instantiation Environments for Java Embedded Software

In general the usage of a bean-based automated instantiation environment depends on three preconditions being satisfied, namely:

- The target application must be built as a collection of components.
- The target application must be written in Java.
- The application components must be encapsulated in Java beans.

In case of Real-Time Java version of AOCS framework are all fulfilled.

6.1 Integration of Simulink's models into the framework

The software framework introduced in the section 2.1 is aimed at the development of embedded control systems. Embedded control systems always include control algorithms developed by control engineers. Control engineers are used to design control algorithms in the Matlab and Simulink computation and simulation environment. A simulation model created in Simulink is stored in a textual file that defines the whole structure of a designed system.

We have built an application that processes the Simulink model file and generates a component that encapsulates the algorithm and that can be directly plugged into an application instantiated from the framework.

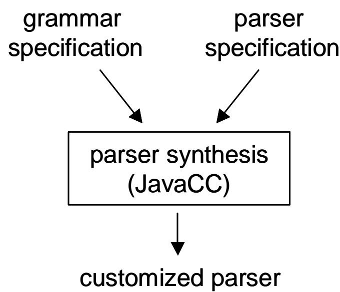

Figure 5. Construction of customized parser

The application converting a Simulink model into a framework component was designed to be compatible with future changes of syntax and semantics in Matlab/Simulink environment. We have used a third party parser generator called JavaCC that allows to automatically generate a customized parser for a user defined grammar [6]. The construction process for the customized parser is shown in Figure 5.

The customized parser requires as an input argument the model file generated by Simulink. First the Simulink file describing the control system is parsed, and then the customized component code is generated. The entire procedure of a Simulink model conversion into pluggable component is illustrated in Figure 6.

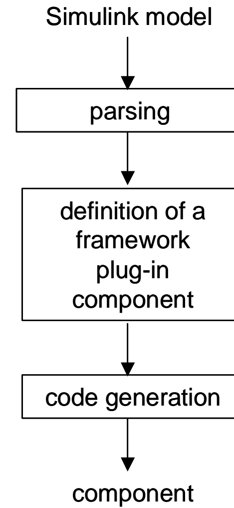

Figure 6. The process of a conversion from a Simulink model into the framework component.

7 CONCLUSION

There are four conclusions to this paper. The first conclusions is that standard Java is at present unsuitable for real-time systems primarily because of its lack of heritage in mission- and safety-critical applications and because it does not support real-time programming.

The second conclusion is based on a market survey of suppliers of Java implementations that showed that RT-compliant versions of Java capable of supporting the needs of real-time applications are already available as commercial products and that they will soon be followed by versions that comply with formal extensions of the language to cover real-time needs.

The third conclusion is supported by the experience of porting the AOCS Framework to a RT version of Java. Within the limitations of the selected platform – Esmertec's Jbed – the porting exercise was very successful. As expected, the chief benefit of the switch to Java lay in the possibility of

making the framework code compatible with multi-threaded operation.

The last conclusion is that the Java language is the natural implementation vehicle for software frameworks and is attractive because of its safety features. It underlies the JavaBeans standard that offers environments where the instantiation process for software frameworks can be automatized.

On the whole, our assessment is that, once the expected implementations of RT Java come on the market, implementation of real-time applications in Java poses no major problem and that Java has the potential of replacing Ada as the language of choice for this class of implementations.

REFERENCES

1. Aicas Home Page, http://www.aicas.com/index.html

2. aJile Home Page, http://www.ajile.com

3. AOCS Framework Project, http://www.aut.ee.ethz.ch/~pasetti/AocsFramework/index.html

4. Esmertec Home Page, http://www.esmertec.com/p.html

5. Gamma E, et al., Design Patterns – Elements of Reusable Object Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995

6. JavaCC Home Page, http://www.webgain.com/products/java_cc/

7. J Consortium Home Page, http://www.j-consortium.org/

8. Newmonics Perc Home Page, http://www.newmonics.com/

9. NIST Special Publication, Requirements for Real-time Extensions for the Java Platform: Report from the Requirement Group for Real-time, http://www.nist.gov/itl/div897/ctg/real-time/

10. Pasetti A., Software Frameworks and Embedded Control Systems, LNCS Series, Springer-Verlag, 2001

11. Real-Time Java Project Web Site, http://www.aut.ee.ethz.ch/~pasetti/RealTimeJavaFramework/index.html

12. Real-Time for Java Expert Group (to be published) The Real-Time Specification for Java. Addison-Wesley. Draft available from: http://www.rtj.org/

13. Sun Microsystems, JavaBeans Specification, java.sun.com/products/javabeans/docs/spec.html